【版权所有、侵权必究】

Spring 3.x 权威开发指南：实施 Java EE 6 的利器

罗时飞 著

http://www.open-v.com

2011 年 8 月 31 日

【版权所有、侵权必究】

目 录

序

从 2003 年开始，开源 Spring 一直在同 Java EE 携手走来。Spring 2.0 之前（包括 1.x、2.0 版本）的版本一直在跟进 J2EE 1.4-的发展，而 Spring 2.5 跟进 Java EE 5，Spring 3.0 开始跟进 Java EE 6 的发展，而 Spring 3.1 全面跟进 Java EE 6，包括 Java SE 7 集成支持。当然，这种跟进的深度和广度远远超越了 Java EE 本身。

Java EE 被企业生产环境广泛采纳，这在很大程度上要归功于这样几方面的因素。其一，Java 本身的跨平台能力，即其可移植性强。其二，Java EE 服务器内置了大量的重要容器服务，比如事务服务、JNDI 服务及连接池服务，这些服务可供企业应用选用。其三，开发者可以基于 Java EE API 研发企业应用，并部署到企业生产环境中。然而，Java EE 暴露给开发者的客户视图存在重大缺陷，尤其是应用同 Java EE 容器打交道的过程很复杂，加上直接基于 Java EE API 研发企业应用的生产效率低下（指研发效率，不是运行效率），并且非常容易出问题，比如数据库连接泄漏。自 Spring 诞生以来，这些问题已经成为了历史。Spring 针对 Java EE 容器服务及 Java EE API 提供了抽象和集成支持，进而得到开发者的广泛拥护。

为更好地针对 Java EE 容器服务及 Java EE API 提供抽象和集成支持，Spring 提供了由控制反转容器和 AOP（面向切面编程）组成的元框架。注意，这一元框架能够被使用到任意场合，而不只是在 Java EE 容器中。

值得提醒的是，Spring 并不能取代 Java EE，但借助 Spring 能降低实施 Java EE 的门槛，并加快采纳 Java EE 的速度。最终，Spring 还能够确保企业应用的质量是一流的。因此，Java EE 是前提，没有 Java EE 的发展，Spring 就不会出现。与此同时，Spring 的出现使得 Java EE 能够得到更好的发展，比如 Java EE 5、6 便吸取了大量的开源经验和成果（其中包括了 Spring 的部分特性），并融入到 Java EE 规范中。

现如今，Spring 已经进入到 3.1 时代。此时进入我们视野的不再局限于 Spring 3.1 框架本身，而是涉及面极广的框架集合，比如面向 Web 服务的 Spring Web Services、面向页面流的 Spring Web Flow、面向集成 Adobe BlazeDS（Flex）的 Spring BlazeDS Integration、面向 ETL 领域的 Spring Batch、面向 Android 的 Spring Android 等。

自从 Spring 诞生的那天起，我们便一直在基于它交付各种类型的企业级 Java 应用，并积累了大量的经验、教训。为全面跟进 Spring 3.1 时代，并记录各种宝贵的经验教训，我便萌发了此书的写作工作，这是非常有意思的大挑战。虽然我们是围绕 Spring 3.1 展开全书写

作工作的，但有关 Java EE 6 的各种背景、实践及技巧都会被深入讨论到。因此，我更希望开发者能够把它看成是一本同时探讨 Java EE 6 和 Spring 3.1 的著作。

如果我们的图书对您有帮助，或者愿意支持图书的持续写作，则可以通过支付宝支持我们，支付宝帐号是：openvcube@gmail.com。让我们一起做得更好！

当然，Spring 涉及的知识面非常广，加上我们经验有限，书中难免出现错误，还望同行批评指正，并提出各种宝贵写作建议。

罗时飞

E_mail：openvcube@gmail.com

2011 年于广州

前言

本书将对 Spring 3.1 及 Java EE 6 进行全方位探索。到目前为止，Spring 3.1 主要提供了三方面的内容：DI 容器、AOP 支持、Java EE 服务抽象及集成。我们的探索旅程正是围绕它们展开的。

我们将各章的主体内容安排如下。

- 第 1 章，借助 Spring 3.1 实施 Java EE 6。
- 第 2 章，控制反转容器。控制反转容器和面向切面编程是 Spring 的核心内容，它们构成了 Spring 本身的元框架。本章探讨 IoC，下章探讨 AOP。
- 第 3 章，面向切面编程。AOP 的出现使得“强入侵性”一词可以寿终正寝了。而且，可喜的是，Spring 内置了多种不同风格的 AOP 实现。
- 第 4 章，DAO 层集成支持。本章内容将围绕 JDBC、事务抽象、集成测试展开论述。
- 第 5 章，Hibernate、JPA 集成。
- 第 6 章，O/X Mapping 集成支持。Spring 3.0 新引入的重要技术，其最初来自于 Spring Web Services 项目。
- 第 7 章，集成 Java EE 其他容器服务。主要阐述服务层相关集成技术，比如 JNDI、EJB 3.1、JMS、JavaMail、JMX 等。
- 第 8 章，Web 层集成支持。
- 第 9 章，高级 Spring 3.1 特性。
- 第 10 章，附录 A，安装及使用 SpringSource Tool Suite。
- 第 11 章，附录 B，Spring 3.1 内置的命名空间。
- 第 12 章，附录 C，Spring Web Services。
- 第 13 章，附录 D，Spring Web Flow。
- 第 14 章，附录 E，Spring BlazeDS Integration。
- 第 15 章，附录 F，Spring Roo。
- 第 16 章，附录 G，相关资料。

值得注意的是，http://openv-cube.googlecode.com 提供了全书配套代码、脚本的下载，借助如下 SVN 命令能够将它们下载到 D:\springsource\ebooks 位置。

```
D:\springsource>svn co http://openv-cube.googlecode.com/svn/trunk/ ebooks
```

随后，开发者可以使用它们，或在 STS 中导入各自的代码或脚本，并完成各自运行和调试工作。如果需要不定期更新它们，则可借助如下 SVN 命令。

```
D:\springsource\ebooks>svn update
```

任何问题，可以同我们取得联系，谢谢！我们特提供了 QQ 群（106813165），以探讨同本书相关的技术问题。

1　借助Spring 3.1 实施Java EE 6

Spring 在 Java EE 领域占据着不可或缺的位置。Java EE 6 将同 Spring 3 并肩战斗，并将企业级 Java 推向另一个高度。

1.1　Java EE 6 编程模型讨论

走过 10 年后，Java EE 已经奔向了“6”字头时代，这在企业计算领域是一种奇迹。然而，一路走来，参与 Java EE 的最广泛群体并没有感受到 Java EE 的研发优势，尤其是它暴露的编程模型。这一最广泛群体正是开发人员，你我都应该属于这一群体。

现如今，“敏捷”一词到处可见，我们就从它开始讨论 Java EE 6 是否已经足够敏捷。

1.1.1　Java EE 6 够敏捷，No！

打开 Java EE 6 规范（http://java.sun.com/javaee/technologies/javaee6.jsp），我们能够看到，这次的改进非常多。这其中有如下几方面内容尤为突出（从暴露给开发者的客户视图阐述，即 Java EE 编程模型）。

其一，加强了模块化、可扩展能力，比如 Servlet 3.0 引入了 Web Fragment 等。

其二，注解技术（Annotation）的使用到处可见。

其三，Java EE API 进行了最广泛的升级，几乎涉及到 Java EE 的各个方面，并引入了 REST 架构（Representational State Transfer，表述性状态转移）。

其四，借助 JSR330 规范（Dependency Injection for Java）、JSR299 规范（Contexts and Dependency Injection for Java，CDI）加强了依赖注入能力。

其五，使用 JSR303 规范（Bean Validation）完成各种复杂校验工作，这是一项非常实用的技术。

从这一列表能够看出，Java EE 6 编程模型确实改进不少，而且在很多细节上下了不少功夫。然而，如下几方面的问题确实也需要开发者仔细考虑一下。

其一，完全借助 Annotation 描述整个企业应用的所有元数据是否合适？毕竟，这容易让开发者“只见树木，不见森林”。

其二，直接借助 Java EE API 是否能够很方便实施各种代码级的测试工作，尤其是集成

测试？

其三，直接采纳 Java EE API 研发企业应用的研发效率如何？

其四，Java EE 6 应用的可移植性能够得到保证吗？

其五，依据以往经验，我们开发的企业应用是否启用了大量开源技术（包括第三方商业技术），比如 Struts、Hibernate、Ehcache、Quartz、Apache DBCP、AspectJ？难道要我们编写集成开源技术的各种代码？

其六，OSGi 将会在不远的将来得到流行，Java EE 应用是否能够顺利移植到新的 OSGi 环境中？虽然不少应用服务器厂商都拥抱了 OSGi，但这只是它们内部行为，它们并没有将 OSGi 行为暴露给开发者。

其七，其他未列思考，留给开发者。

在面对这些问题时，Java EE 6 编程模型不会给开发者“敏捷”的印象，但这绝不会影响到 Java EE 在企业、软件厂商、开发者心中的位置，因为这是不同层次的问题。为了做得更好，我们需要提升敏捷能力。

1.1.2 盘旋于具体与抽象之间

Java EE 6 作为大多数厂商一起制定的重要技术标准、企业运算平台，为保证它的通用性，其内容必然是较抽象的。然而，在借助 Java EE 6 日常研发工作期间，开发者必须解决各种具体问题，比如日志管理、安全性管理、任务调度、应用的可移植性、快速实施测试工作（包括 TDD 的实施）等。

我们注意到，在企业应用领域流行多年的 Spring（http://www.springsource.org/）能够同时摆正具体与抽象的关系，比如对 Java EE 进行了多种层面的抽象和集成。有关 Spring 是如何抽象和集成 Java EE 方面的内容，本章并不想深入阐述，开发者将会在本书的后续章节中一一阅读到它们。开发者暂时只需要记住一点，Spring 巧妙地解决了上节列举出的若干问题，又没有降低 Java EE 6 的企业级能力。Spring 不仅继承了 Java EE 6 的优秀特质，还加强了它们。

在拥抱 Java EE 的前提下，Spring 尽可能抽象及集成各种主流技术，并暴露统一、简单的客户视图给开发者，即 Spring 编程模型。因此，“架构级”的 Spring 够敏捷！我们通常将 Spring 看成是架构级的框架，因为它能够支撑起整个企业应用，应用的各个组成部分都围绕它打转。

1.2 挖掘Spring 3.1 的架构价值

为保证 Spring 3.1 平台的品质，SpringSource 团队作出了许多关键决定，并推出了堪称端到端的一整套产品。比如，面向运行时的 Spring Framework、面向开发者的 SpringSource Tool Suite、面向 Android 的 Spring Android、面向安全性领域的 Spring Security 等。它们无不散发出 Spring 3.1 的架构价值。

1.2.1 精耕细作于Java EE 6 平台

过去的多年中，Spring 一直在同 Java EE 并肩战斗。Spring 3.1 将全面拥抱 Java EE 6 平台。Java EE 作为广泛使用的企业计算平台，其企业运算能力不言而喻。一直以来，Java EE 暴露给开发者的客户视图不够理想，比如研发效率低、实施测试工作较困难等。Spring 看到这些弊端，并试图解决它们。结果是喜人的，毕竟 Spring 已经得到大多数企业、软件厂商、开发者的拥护。

Java EE 的每次重大升级，Spring 总是会积极跟进。同 Java EE 5 相比，Java EE 6 改进的内容非常多。比如，将注解技术（Annotation）应用到 Java EE 的每个角落、支持 REST 架构、Web 层技术进行了全面升级（Servlet 3.0、JSF 2.0）、实用和细粒度的依赖注入（JSR330 和 JSR299）、Bean Validation（JSR303）、EJB 3.1 的推出、JPA 2.0 等。全面跟进它们，成了 Spring 3.0 的主旋律之一。

除了跟进 Java EE 6 外，Spring 3.1 新增或加强的其他内容非常多，比如要求 Java SE 5.0+、引入 Spring 表达式语言（Expression Language，简称 SpEL）、借助注解定义受管 Bean（@Bean、@Lazy 等）、OXM（即对象到 XML 的映射）集成支持、嵌入式数据库支持等。这些特性在 Spring 3.0 中占据着重要位置。

1.2.2 面向Spring的SpringSource Tool Suite集成开发工具

SpringSource Tool Suite（简称 STS，http://www.springsource.com/products/sts）集成开发工具构建在 WTP 基础之上，它是针对 Spring 开发者而来的。加上，它是 SpringSource 公司官方打造的，而且完全免费。

STS 内置大量 Eclipse 插件，比如 Spring IDE、AspectJ 开发工具（http://eclipse.org/ajdt/，

AJDT）、M2Eclipse（http://m2eclipse.sonatype.org/）、Ant 集成、JUnit 测试、WTP/RCP 支持、OSGi 集成支持等。借助 STS，开发者能够以可视化或图形化方式完成 Spring 配置文件的编写，比如 Spring Web Flow 页面流程、Spring Batch 作业。

在 STS 内部存在一 STS Dashboard（仪表盘），透过它，开发者能够积极跟进 Spring 的最新发展趋势、Spring 教程、知识库等。

可以看出，STS 为 Spring（OSGi）应用的开发提供了一端到端的解决方案。附录 A 完整介绍了 STS 的安装及使用。本书也全面启用了 STS，以支持 Spring 应用的研发。

1.2.3 全面拥抱OSGi 4.2

OSGi 是目前最具潜力，能够将 Java EE 带入下一个黄金 10 年的重要技术。一直以来，开发者及企业用户对 Java EE 的模块化和动态能力并不满意，这些都是 Java EE 的软肋。尽管 Java EE 6 的模块化能力加强了不少，比如 Servlet 3.0 引入的 Web Fragment、但还是不够彻底。相反，OSGi 的这些能力很强，但其企业级能力稍差。因此，业界在努力将 OSGi 引入到 Java EE 领域，或者说在集成它们的各自优势。

Spring 看到这一趋势，并推出了 Spring DM 项目（http://www.springsource.org/osgi），还参与 OSGi 4.2 规范的制定工作。与此同时，各种 Spring 项目的正式发布版都遵循 OSGi 的约定，各自的 JAR 包都是标准的 OSGi Bundle，而且项目内部结构进行了全面的梳理，使得各 Spring 项目更模块化、遵循 OSGi 规范，以最大程度享用 OSGi 带来的巨大价值。比如，我们熟知的 Spring Framework 和 Spring Security 全面拥抱了 OSGi。实际上，所有的 Spring 项目都是如此。

值得注意的是，Spring DM 已经成为了 Gemini（http://www.eclipse.org/gemini/）的子项目，并被更名，即 Gemini Blueprint。SpringSource 原先推出的、基于 OSGi 的 dm Server 也捐献给了 Eclipse，即 Virgo（http://www.eclipse.org/virgo/）项目。

有关 OSGi 4.2、Gemini、Virgo 的更多资讯，开发者可参考我们写作的《未来 10 年：OSGi、Gemini、Virgo》电子图书。

1.2.4 开发者决定一切

在采纳 Spring 3.1 平台研发企业应用期间，开发者会发现，他们可以随心所欲。基于

Spring 的基代码能够同时运行在不同的企业环境中，比如 Java SE、Java EE、OSGi，而且在这些差异化的环境中表现出的行为是惊人的一致。

Spring 从来就不会为开发者做决定，而只是尽可能给出多种选择。比如，为实现同 RDBMS 的交互，开发者几乎可借助 Spring 敏捷使用到各种技术，从 JDBC（JdbcTemplate），到 O/R Mapping 技术（JPA、Hibernate、iBATIS、JDO）。

因此，开发者能够决定一切，Spring 从来也是支持的，因为 Spring 3.1 平台来自于开发者，并服务于开发者。

1.3 下载及构建Spring 3.1

本节将围绕 Spring 3.1 的下载和构建工作展开论述。

1.3.1 下载Spring 3.1 正式发布版

现如今，为下载各种 Spring 项目的官方发布版，比如 Spring Framework、Spring Security、Spring Web Services，开发者需要到 http://www.springsource.com/download/community 网址找寻它们，图 1-1 给出了对应的界面截图。

- Spring Framework
  - Latest Development release: 3.1.0.M2
    spring-framework-3.1.0.M2-with-docs.zip (sha1) 48.1 MB
    spring-framework-3.1.0.M2.zip (sha1) 26.5 MB
  - Latest GA release: 3.0.6.RELEASE
  - More >>
- Spring Security
- Spring Web Flow
- Spring Web Services
- Spring Dynamic Modules
- Spring Integration
- Spring Batch
- Spring Batch Admin

图 1-1 下载 Spring 3 的操作界面

当前，最新的 Spring Framework 发布版是 3.1.0.M2，它对应 2 个不同的下载包。其中，spring-framework-3.1.0.M2.zip 仅仅持有 Spring 源码和 Spring JAR 包集合，而 spring-framework-3.1.0.M2-with-docs.zip 新增了各种文档，比如 API 规范、Spring Framework Reference Documentation。下载并解压它们后，透过图 1-2 能够了解到，开发者将需要根据自身的情况，而启用其中的若干 JAR 包。

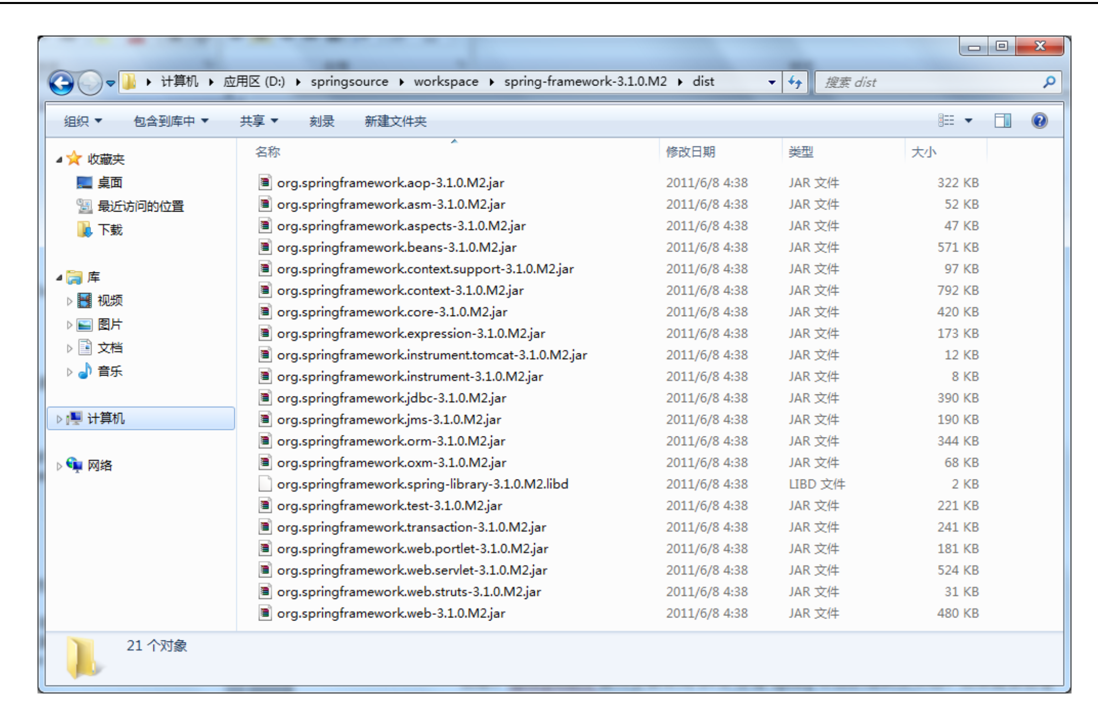


图 1-2　解压后的 Spring Framework

1.3.2 基于SVN库持续构建Spring源码

目前，SpringSource 官方主要采用 SVN 存储 Spring 项目的源码及文档。如果需要或者感兴趣，开发者可以持续从 SVN 库（https://src.springsource.org/svn/spring-framework/trunk/）获得最新的 Spring 项目快照，并构建出最新版本的 Spring。为达到这一目的，开发者可依据如下给出的操作步骤进行。

其一，借助 svn 命令行或 Subclipse，从 SVN 库将 Spring 源码下载到开发者机器中。这里假定将 Spring 源码下载到 D:\springsource\workspace\spring-framework 位置。下面展示了 svn 命令行的使用。

```
svn co https://src.springsource.org/svn/spring-framework/trunk/ spring-framework
```

其二，于 D:\springsource\workspace\spring-framework\build-spring-framework 目录运行 ant 命令行，即构建 Spring 源码。整个构建过程可能会持续 20 分钟左右，时间主要取决于机器及网络状况。注意，在运行它之前，要设置好 Ant 相关内容，尤其是要设置好 ANT_OPTS 环境变量，使得宿主 Ant 的 JVM 有足够的内存支撑构建工作。下面给出了针对 Ant 设定的环境变量示例。或者，开发者可参考 build-spring-framework 目录中的 readme.txt 说明文件。

```
ANT_HOME=D:\apache-ant-1.8.2
ANT_OPTS=-Xms256m -Xmx1024m -XX:MaxPermSize=256m
```

其三，细心的开发者会发现，spring-framework 目录由多个 Eclipse 工程构成，比如 AOP（org.springframework.aop）、事务集成（org.springframework.transaction）、O/R Mapping 集成（org.springframework.orm）等。如果开发者需要在 Eclipse 中操控这一 Spring 项目快照，则在将这些 Eclipse 工程导入之前，要在 Eclipse（STS）中设置“IVY_CACHE”类路径变量，其指向“D:/springsource/workspace/spring-framework/ivy-cache/repository”位置，图 1-3 展示了这一设置。可以看出，Spring 源码是基于 Apache Ivy 组织的，上述 ant 命令行执行期间，会从远程 Ivy 库下载所需的各种第三方 JAR 包，并存储到 ivy-cache 位置。

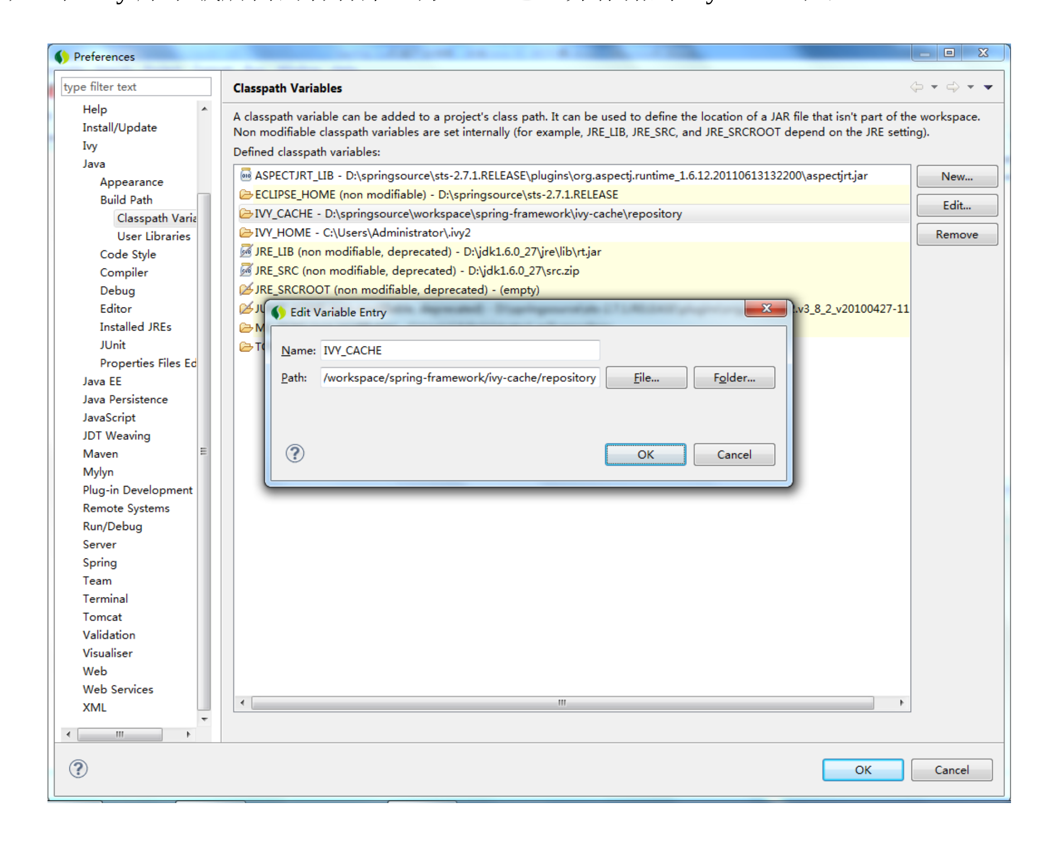


图 1-3　设置 IVY_CACHE 类路径变量

总之，开发者可以根据自身的不同情况，来合理构建或编译直接从 SVN 版本库获得的 Spring 源码。

1.4　小结

本章大致描述了 Java EE 6 的缺陷、借助 Spring 3.1 平台实施 Java EE 6 的重要意义、以及 Spring 3.1 的下载和构建。

从下章开始，我们将逐步渗透到 Spring 3.1 平台的各个角落中。

2 控制反转容器

DI 容器（或称之控制反转容器，Inversion of Control，IoC），负责管理宿主其中的 Spring 受管 Bean（POJO），比如生命周期管理、协作者、事件分发、资源查找等。与此同时，Spring 内置了一流的 AOP 技术实现，并同 AspectJ 进行了无缝集成。

Spring 提供的 IoC 容器和 AOP 技术实现构成了 Spring 的核心内容，它们是 Spring 元框架，其中 Spring 内置的 Java EE 服务抽象和集成便是架构在这一元框架基础上的。

本章内容将围绕 DI 容器展开，下章将围绕 AOP 技术实现展开。

2.1 DI及Spring DI概述

类似于 EJB 容器管理 EJB 组件一样，Spring DI 容器负责管理宿主在其中的受管 Bean，或者称之为受管 POJO、Spring Bean 等。就目前来看，Spring 内置了两种基础 DI 容器，即 BeanFactory 和 ApplicationContext，它们间的关系见图 2-1。

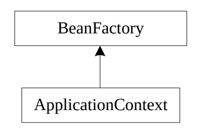


图 2-1 两种基础 DI 容器

下面来分别研究这两种 DI 容器。

2.1.1 面向Java ME/Java SE的BeanFactory

BeanFactory 主要用在内存、CPU 资源受限场合，比如 Applet、手持设备等。它内置了最基础的 DI 功能，比如配置框架、基础功能。开发者经常会使用到 Spring 内置的图 2-2 实现，即 XmlBeanFactory。

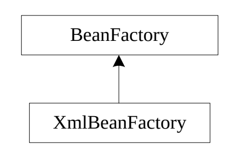


图 2-2 XmlBeanFactory

在企业级计算环境，开发者往往要使用 ApplicationContext，而 BeanFactory 是不能够胜

任的。

2.1.2 面向Java EE的ApplicationContext

在 BeanFactory 基础上，ApplicationContext 提供了大量面向企业计算所需的特性集合，比如消息资源的国际化（i18n）处理、简化同 Spring AOP 的集成、内置事件支持、针对 Web 应用提供了诸多便利、资源操控等。开发者会经常在各种场合使用到图 2-3 列举出的 DI 实现。比如，面向 Web 应用的 XmlWebApplicationContext 容器、基于注解存储 DI 元数据的 AnnotationConfigApplicationContext 容器、适合于各种场景的 ClassPathXmlApplicationContext 和 FileSystemXmlApplicationContext、面向 Portal 应用的 XmlPortletApplicationContext。

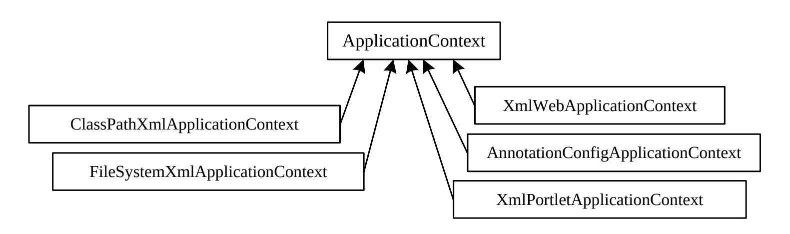


图 2-3 常见 ApplicationContext 实现

除此之外，Spring 还内置了一些其它类型的 ApplicationContext 实现，比如面向 JCA 资源适配器环境的 ResourceAdapterApplicationContext。

值得注意的是，ApplicationContext 继承了 BeanFactory 的所有特性。无论如何，本书的所有场合将围绕这两种类型的 DI 容器展开阐述。

2.2 多种依赖注入方式

Spring DI 容器支持多种不同的依赖注入类型，比如设值注入、构建器注入、属性注入、方法注入等。接下来，我们将围绕 Eclipse iocdemo 项目阐述它们。

2.2.1 设值注入

设值（setter）注入，指通过调用 setter 方法，从而建立起对象间的依赖关系。比如，BankSecurityServiceImpl 实现类定义了如下 setBankSecurityDao()设值方法。

```
public class BankSecurityServiceImpl implements IBankSecurityService {

    private IBankSecurityDao bankSecurityDao;
```

```
    public void setBankSecurityDao(IBankSecurityDao bankSecurityDao) {
        this.bankSecurityDao = bankSecurityDao;
    }

...
```

借助 Spring DI 容器，开发者可将 IBankSecurityDao 对象提供给 BankSecurityServiceImpl。下面给出了 beanfactory.xml 中的全部内容，它们承载了 Spring DI 元数据。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="bankSecurityService"
        class="com.openv.spring3x.ioc.BankSecurityServiceImpl">
        <property name="bankSecurityDao" ref="bankSecurityDao"/>
    </bean>

    <bean id="bankSecurityDao"
        class="com.openv.spring3x.ioc.BankSecurityDaoImpl">
    </bean>

</beans>
```

从上述 XML 配置信息能够看出，对象间的依赖关系可以外在化进行管理。下面展示了 BeanFactoryDemo 示例应用，它加载了 beanfactory.xml DI 容器，并通过 BeanFactory 暴露的 getBean()方法访问到 IBankSecurityService 服务。

```
//从 classpath 路径上装载 XML 配置信息
Resource resource = new ClassPathResource("beanfactory.xml");

//实例化 IoC 容器，此时，容器并未实例化 beanfactory.xml 所定义的各个受管 Bean
BeanFactory factory = new XmlBeanFactory(resource);

//获得受管 Bean
IBankSecurityService bankSecurityService =
    (IBankSecurityService) factory.getBean("bankSecurityService");

bankSecurityService.bankToSecurity(2000.00);
bankSecurityService.securityToBank(2000.00);
```

☺，我们并没有手工构造任何对象，BankSecurityServiceImpl 和 BankSecurityDaoImpl 实例都是由 BeanFactory 构造的，而且它们的依赖关系设置也是由这一 DI 容器完成的。我们所做的事情，只是告诉了 BeanFactory 这些对象间的协作关系而已。这正是 DI 的优势所在，“Don’t call me, I’ll call you!”。

懒惰的 BeanFactory

初始化 BeanFactory 实例后，IoC 容器并不会预先实例化配置文件中已声明的各个受管 POJO。只有应用在使用到对应的 POJO 时，Spring 才会实例化使用到的受管 Bean。

对于 ApplicationContext 而言，则不同。一旦构造完 ApplicationContext 对象，IoC 容器便会预先实例化配置文件中已声明的各个受管 POJO。这是同 BeanFactory 的区别之一。

开发者可以去试图运行 BeanFactoryDemo 示例应用，并仔细分析一下发生的一切。如果抱怨 beanfactory.xml XML 难于编写和维护，则可借助 STS 内置的 Spring IDE 支持。附录 A 和附录 B 有详细的背景知识可供开发者参考。或者，AnnotationConfigApplicationContext 容器可干掉您的烦恼，因为它可完全借助注解承载 DI 元数据。

2.2.2 构建器注入

构建器注入，指往构建器传入若干参数而完成的依赖注入，传入的各个参数都是受管 Bean 依赖的对象，这些对象间构成了协作关系。比如，下面提供了一构建器。

```
public BankSecurityServiceImpl(IBankSecurityDao bankSecurityDao) {
    this.bankSecurityDao = bankSecurityDao;
}
```

借助<constructor-arg/>子元素，我们能够将 IBankSecurityDao 对象通过构建器注入到 BankSecurityServiceImpl 中，配置示例如下。

```
<bean id="bankSecurityService"
    class="com.openv.spring3x.ioc.BankSecurityServiceImpl">
    <constructor-arg ref="bankSecurityDao"/>
</bean>
```

如果构建器持有多个参数，则可同时使用多个<constructor-arg/>子元素，如下示例配置。

```
<bean id="bankSecurityService"
    class="com.openv.spring3x.ioc.BankSecurityServiceImpl">
    <constructor-arg ref="bankSecurityDao"/>
    <constructor-arg value="null"/>
</bean>
```

或者，开发者可依据协作者的类型或位置信息来给出 DI 元数据，配置示例如下。

```
<bean id="bankSecurityService"
    class="com.openv.spring3x.ioc.BankSecurityServiceImpl">
    <constructor-arg index="0" ref="bankSecurityDao"/>
    <constructor-arg type="java.util.Properties" value="null"/>
</bean>
```

开发者能够决定一切，根据自身需要灵活使用构建器注入吧！

2.2.3 属性注入

在没有提供构建器和设置方法的前提下，借助属性注入，我们同样可以将协作者注入进来。比如，借助@Autowired、@Resource、@EJB 等注解。这里介绍@Autowired 的使用，下面给出了使用示例，摘自 BankSecurityServiceImpl。

```
@Autowired
private IBankSecurityDao bankSecurityDao;
```

为激活这一注解，我们需要在 DI 容器中配置 AutowiredAnnotationBeanPostProcessor 对象，具体示例如下。此时，我们没有在 XML 配置信息中给出对象间的协作关系。

```
<bean id="bankSecurityService"
    class="com.openv.spring3x.ioc.BankSecurityServiceImpl">
</bean>

<bean id="bankSecurityDao"
    class="com.openv.spring3x.ioc.BankSecurityDaoImpl">
</bean>

<bean class="org.springframework.beans.factory.annotation
                 .AutowiredAnnotationBeanPostProcessor"/>
```

ApplicationContextDemo 示例应用加载了这一 DI 容器，具体如下。BeanPostProcessor 类型对象需宿主在 ApplicationContext 容器中，因此它启用了 ClassPathXmlApplicationContext。

```
//从 classpath 路径上装载 XML 配置信息
ApplicationContext applicationContext =
    new ClassPathXmlApplicationContext("applicationcontext.xml");

//获得受管 Bean
IBankSecurityService bankSecurityService =
    applicationContext.getBean(IBankSecurityService.class);
```

2.2.4 方法注入

Spring 内置了两种方法注入策略，即查找（Lookup）方法注入和方法替换（Replacement）注入。其中：

查找方法注入，指重载受管 Bean 的（抽象）方法。它会将查找到的其他受管 Bean（通常为原型）替换现有方法的返回结果，从而起到方法注入的效果。在实施查找方法注入时，

开发者需要启用<bean/>中的<lookup-method/>子元素。

方法替换注入，指使用其他受管 Bean 实现的方法替换目标受管 Bean 中的现有方法。在实施方法替换注入时，开发者需要启用<bean/>中的<replaced-method/>子元素。

借助方法注入，能够“优雅”解决应用中单例对原型的引用，从而避免应用同 Spring 的深度耦合。作者认为，方法注入的使用过于复杂，而且不实用。我们完全可以通过改进应用的架构设计，以避免它们的使用。更何况，即使应用同 Spring 耦合也不是太大的问题，因为它是事实上的开发标准。有关方法注入的具体细节这里就不介绍了，如果您感兴趣，则可以参考本书附录列出的参考资料。

2.3 借助Autowiring策略智能注入协作者

所谓 Autowiring 策略，即自动管理受管 Bean 间的关系，开发者不用手工一一指定它们间的协作关系，很显然，这大大降低了 XML 配置文件的维护量。Spring 内置了多种不同的 Autowiring 策略，本节内容将一一介绍它们。

2.3.1 <bean/>元素的autowire属性

开发者借助<ref/>元素或 ref 属性能够显式指定当前受管 Bean 的协作者，比如下面给出了配置示例。

```
<bean id="bankSecurityService"
    class="com.openv.spring3x.ioc.BankSecurityServiceImpl">
    <property name="bankSecurityDao">
        <ref bean="bankSecurityDao"/>
    </property>
</bean>
```

试想，如果待配置的协作者数量惊人，而且 Spring XML 配置文件非常多，则维护它们是非常痛苦的事情。通过启用<bean/>元素内置的 autowire 属性，DI 容器能够自动完成协作者的依赖注入。下面一一列举出了 autowire 属性的取值集合。

表 2-1　autowire 属性取值集合

autowire 属性取值	详细解释
byName	通过属性名完成 autowire 操作。如果某受管 Bean a 含有 b 属性，则 IoC 容器会去所有受管 Bean 中寻找名字（id）为 b 的受管 Bean，并注入到受管 Bean a 的 b 属性中。如果未找到，则不会设置 b 属性的取值
byType	通过判断属性类型完成 autowire 操作。如果某受管 Bean a 中 b 属性的类型（type）为

autowire 属性取值	详细解释
	IBankSecurityDao，则 IoC 容器会去所有受管 Bean 中寻找类型为 IBankSecurityDao 的受管 Bean，并注入到受管 Bean a 的 b 属性中。如果找到多个，则会抛出异常。如果未找到，则不会设置 b 属性的取值
constructor	同 byType，但作用于构建器
no	不启用 autowire 特性，这是 Spring 受管 Bean 的默认行为
default	同<beans/>级别设定的 autowiring 策略

比如，BankSecurityServiceImpl 中定义了如下 setter 方法。

```
public void setBankSecurityDao(IBankSecurityDao bankSecurityDao) {
    this.bankSecurityDao = bankSecurityDao;
}
```

启用如下“byType”策略，我们能够将 BankSecurityDaoImpl 受管 Bean 自动注入进来，即为 BankSecurityServiceImpl 指定协作者。

```
<bean id="bankSecurityService"
    class="com.openv.spring3x.ioc.BankSecurityServiceImpl"
    autowire="byType"/>

<bean id="bankSecurityDao"
    class="com.openv.spring3x.ioc.BankSecurityDaoImpl"/>
```

或者，如果 BankSecurityServiceImpl 内置了如下构建器，则启动“constructor”策略能够完成协作者的智能注入。

```
public BankSecurityServiceImpl(IBankSecurityDao bankSecurityDao) {
    this.bankSecurityDao = bankSecurityDao;
}
```

最后，为启用“byName”策略，协作者的 id 或名字必须同属性名一致。下面给出的配置示例将不能成功将协作者注入到 BankSecurityServiceImpl 受管 Bean（假定它仍然定义了上述 setter 方法）中，因为它暴露的属性名是 bankSecurityDao，而 BankSecurityDaoImpl 受管 Bean 的名字为 bankSecurityDaoImpl。

```
<bean id="bankSecurityService"
    class="com.openv.spring3x.ioc.BankSecurityServiceImpl"
    autowire="byName"/>

<bean id="bankSecurityDaoImpl"
    class="com.openv.spring3x.ioc.BankSecurityDaoImpl"/>
```

2.3.2 基于@Required注解加强协作者管理

开发者是否注意到，启用 autowire 属性期间，我们并不能够确定协作者是否找到。通常，

协作者没有找到时，NPE 异常可能会抛出，但这可能已经是运行期行为。我们是否能够在启动应用期间就将这类问题找出来，并告知开发者，从而尽可能避免生产问题的发生。借助@Required 注解，我们能够加强协作者管理。不过，启用这一注解时，需要单独配置如下受管 Bean 定义，并启用 ApplicationContext 容器。

```
<bean class="org.springframework.beans.factory.annotation
        .RequiredAnnotationBeanPostProcessor"/>
```

或者，激活如下配置。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
            http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:annotation-config/>

</beans>
```

下面展示了@Required 注解的使用。这一注解只能够作用在方法上。

```
@Required
public void setBankSecurityDao(IBankSecurityDao bankSecurityDao) {
    this.bankSecurityDao = bankSecurityDao;
}
```

事实上，@Required 注解只能够确保其作用的方法是否被调用到，它也不能确保协作者是否成功注入。即使将<null/>取值提供给其作用的方法，则也是没问题的，即骗过@Required 注解。下面给出了配置示例，它将<null/>，即 null 值给了 bankSecurityDao 属性。

```
<bean id="bankSecurityService"
    class="com.openv.spring3x.ioc.BankSecurityServiceImpl">
    <property name="bankSecurityDao">
        <null/>
    </property>
</bean>
```

正常情况下，当应用启动时，如果@Required 注解作用的方法被调用到，则说明协作者确实找到了；如果@Required 注解作用的方法没被调用到，则应用是不能够正常启动的，即会有如下类似异常的抛出。

```
Caused by: org.springframework.beans.factory.BeanInitializationException:
    Property 'bankSecurityDao' is required for bean 'bankSecurityService'
```

```
    at org.springframework.beans.factory.annotation
        .RequiredAnnotationBeanPostProcessor
        .postProcessPropertyValues(RequiredAnnotationBeanPostProcessor.java:149)
```

2.3.3 基于@Autowired或@Inject注解的另一Autowiring策略

之前介绍属性注入时，我们已经了解到@Autowired 注解的使用了。为使用@Autowired 或@Inject 注解，开发者需要在 DI 容器中单独配置如下受管 Bean。

```
<bean class="org.springframework.beans.factory.annotation
                .AutowiredAnnotationBeanPostProcessor"/>
```

或者，激活如下配置。

```
<context:annotation-config/>
```

@Inject 注解是 JSR330（javax.inject）引入的，它同@Autowired 注解一样，都可以作用到构建器、属性、方法上。下面展示了这一注解作用到属性上的代码示例。

```
@Inject
private IBankSecurityDao bankSecurityDao;
```

默认时，一旦@Autowired 或@Inject 注解作用到构建器，或属性，或方法上，则协作者必须存在，否则应用不能够正常启动。这有点类似于@Required 注解的功效。即，@Autowired 和@Inject 注解内置了@Required 注解的能力，不过，@Autowired 注解允许开发者不启用这一能力。下面给出了代码示例。默认时，required 属性取值为 true，一旦设置为 false 后，即使没有为 bankSecurityDao 属性提供协作者，应用也能够“正常”启动。

```
@Autowired(required=false)
private IBankSecurityDao bankSecurityDao;
```

2.3.4 借助primary属性或@Qualifier注解细粒度控制Autowiring策略

采纳各种 Autowiring 策略时，当 DI 容器中多个不同受管 Bean 都符合条件，并能够作为其他受管 Bean 的协作者时，到底该将哪个受管 Bean 作为真正的协作者呢？

下面给出了配置示例。abcBankSecurityDao 和 cmbBankSecurityDao 受管 Bean 都能够作为协作者注入到 BankSecurityServiceImpl 受管 Bean 中。由于 abcBankSecurityDao 受管 Bean 启用了 primary 属性，而且其取值为 true，因此这一受管 Bean 会被作为 bankSecurityService 受管 Bean 的协作者。

```
<bean id="bankSecurityService"
    class="com.openv.spring3x.ioc.BankSecurityServiceImpl" autowire="byType"/>

<bean id="abcBankSecurityDao" class="com.openv.spring3x.ioc.BankSecurityDaoImpl"
    primary="true"/>

<bean id="cmbBankSecurityDao" class="com.openv.spring3x.ioc.BankSecurityDaoImpl"/>
```

当多个候选受管 Bean 都符合协作者条件时，只有启用了 primary 属性，并且其取值为 true 的受管 Bean 才有资格注入到其他受管 Bean 中，比如 bankSecurityService 受管 Bean。注意，primary 默认取值为 false。另一方面，当多个候选受管 Bean 都符合协作者条件时，而且大家都没有启用 primary 属性，则不能够正常启动应用，相应的异常类似如下。

```
Caused by: org.springframework.beans.factory.NoSuchBeanDefinitionException:
    No unique bean of type [com.openv.spring3x.ioc.IBankSecurityDao] is defined:
    expected single matching bean but found 2:
        [abcBankSecurityDao, cmbBankSecurityDao]
```

值得注意的是，既然可以借助 primary 属性让某个受管 Bean 作为协作者，我们也可以借助 autowire-candidate 属性让某个受管 Bean 压根不参与到 Autowiring 策略中，配置示例如下。此时，cmbBankSecurityDao 受管 Bean 不会作为其它任何受管 Bean 的协作者。能够看出，当 autowire-candidate 属性取值为 false 时，当前受管 Bean 不会参与到 Autowiring 策略中，其默认取值为 true。

```
<bean id="bankSecurityService"
    class="com.openv.spring3x.ioc.BankSecurityServiceImpl" autowire="byType"/>

<bean id="abcBankSecurityDao" class="com.openv.spring3x.ioc.BankSecurityDaoImpl"
    primary="false"/>

<bean id="cmbBankSecurityDao" class="com.openv.spring3x.ioc.BankSecurityDaoImpl"
    autowire-candidate="false"/>
```

位于 org.springframework.beans.factory.annotation 包的@Qualifier 注解是我们这里需要重点关注的另一对象。它配合@Autowired 注解的使用，能够有序地将不同候选受管 Bean 注入到不同地方。比如，下面给出了代码示例。这里定义了 3 个不同的 IBankSecurityDao 变量。

```
@Autowired
@Qualifier("abc")
private IBankSecurityDao abcBankSecurityDao;

@Autowired
@Qualifier("cmb")
private IBankSecurityDao cmbBankSecurityDao;
```

```
@Autowired
private IBankSecurityDao bankSecurityDao;
```

现要求 3 个不同的 IBankSecurityDao 受管 Bean 分别注入进来，下面给出了配置示例。开发者能够看出，<qualifier/>元素同@Qualifier 注解的默契配合完美地完成了这一任务。对于取值为 cmb 的<qualifier/>元素而言，则其所在的 cmbBankSecurityDao 受管 Bean 将注入到上述 cmbBankSecurityDao 变量中，因为作用到这一变量的@Qualifier 注解的取值也为 cmb。类似地，开发者可以依此类推。

```
<bean id="bankSecurityService"
    class="com.openv.spring3x.ioc.BankSecurityServiceImpl">
</bean>

<bean id="bankSecurityDao"
    class="com.openv.spring3x.ioc.BankSecurityDaoImpl" primary="true">
</bean>

<bean id="cmbBankSecurityDao"
    class="com.openv.spring3x.ioc.BankSecurityDaoImpl">
    <qualifier value="cmb"/>
</bean>

<bean id="abcBankSecurityDao"
    class="com.openv.spring3x.ioc.BankSecurityDaoImpl">
    <qualifier value="abc"/>
</bean>
```

@Qualifier 注解的定义如下。它能够作用到许多地方，比如属性、方法、参数、类、其他注解等。

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER,
    ElementType.TYPE, ElementType.ANNOTATION_TYPE})
@Inherited
@Documented
public @interface Qualifier {

    String value() default "";

}
```

下面展示了将@Qualifier 注解作用到参数的代码示例。

```
@Autowired
private void init(@Qualifier("abc") IBankSecurityDao abcBankSecurityDao,
    @Qualifier("cmb") IBankSecurityDao cmbBankSecurityDao,
        IBankSecurityDao bankSecurityDao){
    this.abcBankSecurityDao = abcBankSecurityDao;
    this.cmbBankSecurityDao = cmbBankSecurityDao;
```

```
    this.bankSecurityDao = bankSecurityDao;
}
```

总之，开发者需要依据自身不同情况，而灵活启用不同的 Autowiring 策略。

2.4 资源操控

2.5 将DI容器宿主到Web容器中

2.6 外在化配置应用参数

2.7 Spring受管Bean的作用范围

2.8 Bean Validation集成支持

2.9 回调接口集合及触发顺序

2.10 <util/>命名空间

2.11 值得重视的若干DI特性

本节内容将针对 Spring 内置的一些 DI 特性进行阐述。

2.11.1depends-on属性

2.11.2别名（Alias）

如同叫别人小名一样，我们也可以对受管 Bean 取“小名”。默认时，我们需要在定义<bean/>元素时指定 id 或 name 属性。由于 id 属性是 XML 标准行为，因此一般建议开发者通过给定 id 属性，以唯一标识受管 Bean。通常，如果 id 和 name 属性都没有指定，Spring 会为当前受管 Bean 生成一默认 id 值，即类的全限定名（FQN）。

某些时候，我们需要借助<alias/>元素，给特定受管 Bean 取别名，比如为了达到同具体<bean/>解耦的目的。下面给出了配置示例，以后无论是通过“bankSecurityDao”，还是通过“bsd”别名查找到的受管 Bean 始终是同一个。

```
<bean id="bankSecurityDao"
    class="com.openv.spring3x.ioc.BankSecurityDaoImpl"/>

<alias name="bankSecurityDao" alias="bsd"/>
```

在<alias/>元素中，name 属性用于指定被取别名的目标受管 Bean 的 id 或 name，而 alias 属性用于指定别名。

2.11.3工厂Bean和工厂方法

2.11.4<p/>命名空间

开发者只需往 Spring DI 配置文件中新增如下粗体内容，<p/>命名空间便可启用了。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:p="http://www.springframework.org/schema/p"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
```

假定存在一<bean/>定义如下。其中，bankSecurityDao 为复杂数据类型，而 level 只是一Integer。

```
<bean id="bankSecurityService"
    class="com.openv.spring3x.ioc.BankSecurityServiceImpl">
    <property name="bankSecurityDao" ref="bankSecurityDao"/>
    <property name="level" value="10"/>
</bean>
```

借助<p/>命名空间，我们可给出如下等效配置。其中，“p:*-ref”用于引用其它受管 Bean，*表示属性名，比如 bankSecurityDao；对于简单属性（数据类型），开发者直接启用属性名即可，即“p:*”。

```
<bean id="bankSecurityService"
    class="com.openv.spring3x.ioc.BankSecurityServiceImpl"
    p:bankSecurityDao-ref="bankSecurityDao" p:level="10"/>
```

<p/>命名空间的使用使得配置文件变得简洁。而且，这一命名较特殊，即它不存在对应的 XSD（XML Schema）。

2.11.5抽象和子Bean

2.12 基于注解（Annotation）方式配置DI容器

2.13 Spring表达式语言（SpEL）支持

2.13.1 核心接口及类

2.13.2 基于API方式使用

2.13.3 基于XML方式使用

2.13.4 基于Annotation注解使用

2.13.5 SpEL语法速查

2.14 回调接口集合及其触发顺序

2.14.1 BeanNameAware回调接口

2.14.2BeanClassLoaderAware回调接口

2.14.3BeanFactoryAware回调接口

2.14.4ResourceLoaderAware回调接口

2.14.5ApplicationEventPublisherAware回调接口

2.14.6MessageSourceAware回调接口

2.14.7ApplicationContextAware回调接口

2.14.8@PostConstruct注解

2.14.9InitializingBean回调接口

2.14.10 <bean/>元素的init-method属性

2.14.11 @PreDestroy注解

2.14.12 DisposableBean回调接口

2.14.13 <bean/>元素的destroy-method属性

2.15 小结

Spring 内置的 DI 容器非常强大，易用性也不错。本章对这一业界使用时间最长、功能最强的控制反转容器进行了全方位研究。各种企业计算环境广泛使用并部署了它。而它的搭档－AOP 技术实现，将在下章现身。

3 面向切面编程

借助面向对象（OO）技术能够解决对象和服务的构造、抽象、复用，然而 OO 技术不能够将企业级服务优雅地引入进来，但是本章介绍的 AOP 能够全面担负这种职责。Spring 内置的 AOP 技术实现和 AspectJ 集成支持为各种企业应用的研发提供了一流的解决方案，几乎可以认为，在未来的 10 年中，很难有其它 AOP 解决方案超越它。

好了，我们还是从实践 AOP 开始。

3.1 AOP及Spring AOP基础

3.1.1 细说AOP

3.1.2 Spring AOP基础概念

3.2 AspectJ 6 初探

3.3 老式Spring AOP

3.4 基于@AspectJ的Spring AOP

3.5 基于<aop:config/>元素的AOP

3.5.1 巧用<aop:include/>元素

3.6 在AspectJ 6 应用中启用@Configurable注解

不难发现，org.springframework.aspects-3.0.1.RELEASE-A.jar 包内置了不少 AspectJ 切面，比如 AnnotationBeanConfigurerAspect（同@Configurable 注解配合使用能实现领域对象的 DI 操作）、AnnotationTransactionAspect（同@Transactional 注解配合使用能智能完成事务操作）等，具体细节可参考这一 JAR 包内置的 META-INF/aop.xml 文件。本节将通过启用 AspectJ 装载期织入（LTW）阐述 AnnotationBeanConfigurerAspect 切面与@Configurable 注解的使用，其它切面将在本书相关章节介绍。

借助 AnnotationBeanConfigurerAspect 切面，并配合@Configurable 注解的使用，我们能够实现领域对象的依赖注入操作。领域对象并不是由 Spring 容器创建的，它们可能由“new”关键字构造而成，也可能由 O/R Mapping 工具创建，或通过其它途径获得。

本节内容将围绕 Eclipse configurabledemo 项目进行。

3.6.1 显式使用AnnotationBeanConfigurerAspect切面

为展示 AnnotationBeanConfigurerAspect 切面的使用，我们准备了如下领域对象（证券帐号），它启用了@Configurable 注解。IBankSecurityService 服务负责银行和券商间资金的

转入转出操作。

```
import org.springframework.beans.factory.annotation.Configurable;

@Configurable
public class SecurityAccount {

    private IBankSecurityService bankSecurityService;

    public void setBankSecurityService(IBankSecurityService bankSecurityService) {
        this.bankSecurityService = bankSecurityService;
    }

    public void toBank(Double money){
        this.bankSecurityService.securityToBank(money);
    }

    public void toSecurity(Double money){
        this.bankSecurityService.bankToSecurity(money);
    }

...
```

接下来，我们将 AnnotationTransactionAspect 切面配置在 config.xml 中，相关内容摘录如下。由于领域对象的特殊性，这里将它的作用范围置为原型。

```
<bean class="org.springframework.beans.factory.aspectj.
                 AnnotationBeanConfigurerAspect" factory-method="aspectOf"/>

<bean scope="prototype" class="com.openv.spring3x.aop.SecurityAccount"
     p:bankSecurityService-ref="bankSecurityService"/>

<bean id="bankSecurityService"
              class="com.openv.spring3x.aop.BankSecurityServiceImpl"/>
```

ConfigurableDemo 应用激活了 config.xml，示例代码如下。它手工创建了证券帐号，并调用了相关的转帐操作（toBank()和 toSecurity()）。

```
//启用 Spring DI，并完成 AspectJ 6 切面的配置工作，比如将 IoC 容器暴露给切面
new ClassPathXmlApplicationContext("config.xml");

log.info("即将构建领域对象");

//很多时候，Hibernate/JPA/应用代码会负责创建领域对象
SecurityAccount securityAccount = new SecurityAccount();

log.info("AspectJ 6 已经完成了领域对象的配置工作");

//没有显式设置 securityAccount 领域对象的协作者，但 IBankSecurityService 服务确实不再是 null
securityAccount.toBank(10000);
securityAccount.toSecurity(10000);
```

运行上述应用前，开发者需要设置如下 JVM 参数，从而激活 AspectJ 6 的 LTW 行为。

```
-javaagent:../../lib/com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar
```

图 3-1 展示了 STS 中是如何设置这一 JVM 参数的。

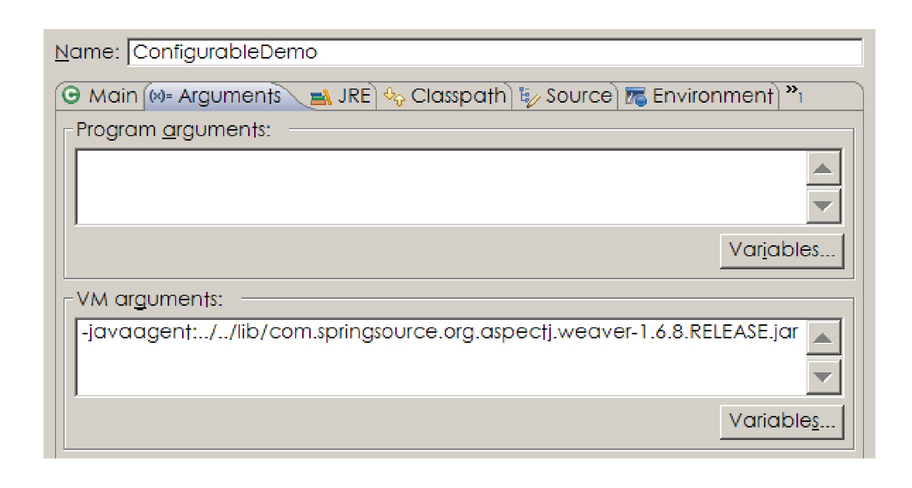


图 3-1　激活 AspectJ 6 的 LTW 行为

应用被执行后，控制台将输出如下类似日志。我们可以推测出，AspectJ 从 config.xml 容器中找到 SecurityAccount 对象，并用它动态替换掉代码中手工创建的 SecurityAccount 对象，从而避免了 NPE 异常的抛出，因为 IBankSecurityService 服务没有机会出现在手工创建的 SecurityAccount 对象中。

```
[INFO] 2009-12-20 00:57:32,781 com.openv.spring3x.aop.ConfigurableDemo - 即将构建领域对象
[INFO] 2009-12-20 00:57:32,828 com.openv.spring3x.aop.ConfigurableDemo - AspectJ 6 已经完成了领域对象的配置工作
[INFO] 2009-12-20 00:57:32,828 com.openv.spring3x.aop.BankSecurityServiceImpl - 即将从券商转出 10000.0 元到银行
[INFO] 2009-12-20 00:57:32,828 com.openv.spring3x.aop.BankSecurityServiceImpl - 即将从银行转入 10000.0 元到券商
```

3.6.2 阐述@Configurable注解

Spring 引入的@Configurable 注解只能够作用于类，通常是领域对象本身，而不能将它运用到方法或其它地方。被应用了这一注解的类将参与到 AspectJ LTW 行为中，并完成协作者的智能注入工作。下面展示了@Configurable 注解暴露的所有成员，通过它们能够自定义若干行为，比如领域对象对应受管 Bean 的查找策略、协作者的注入策略等。

```
/**
 * 领域对象对应受管 Bean 的名字，映射到 Spring DI 容器
 */
```

```
String value() default "";

/**
 * 启用不同 autowiring 策略以注入协作者
 */
Autowire autowire() default Autowire.NO;

/**
 * 是否启用依赖性检查，即检查协作者
 */
boolean dependencyCheck() default false;

/**
 * 领域对象构建前是否注入协作者
 */
boolean preConstruction() default false;
```

比如，value 属性用来指定领域对象对应受管 Bean 的名字，下面给出了配置示例。默认时，AnnotationBeanConfigurerAspect 切面会根据领域对象的全限定名（FQN）查找到对应的受管 Bean 名字，比如"com.openv.spring3x.aop.SecurityAccount"。如果受管 Bean 的定义中未出现 id 或 name 属性，则类的 FQN 便被当作领域对象对应受管 Bean 的名字。

```
@Configurable(value="com.openv.spring3x.aop.SecurityAccount")
```

或者，开发者可以提供如下简写形式。

```
@Configurable("com.openv.spring3x.aop.SecurityAccount")
```

另外，autowire 属性用来指定领域对象注入协作者的 Autowiring 策略。在展示这一属性的使用前，我们得先调整 config.xml 配置文件，将 SecurityAccount 对 IBankSecurityService 服务的静态引用去掉，configautowiring.xml 持有调整后的配置，相关内容摘录如下。

```
<bean scope="prototype" class="com.openv.spring3x.aop.SecurityAccount"/>

<bean class="com.openv.spring3x.aop.BankSecurityServiceImpl"/>
```

如果不启用 Autowiring 策略，则 SecurityAccount 的 IBankSecurityService 服务将永远为空，下面给出了配置示例。

```
@Configurable(autowire=Autowire.BY_TYPE)
```

或者，开发者可在定义 SecurityAccount 时启用<bean/>元素暴露的 autowire 属性，以达到同样的效果，配置示例如下。

```
<bean scope="prototype" class="com.openv.spring3x.aop.SecurityAccount"
    autowire="byType"/>
```

再来研究@Configurable 注解暴露的 dependencyCheck 属性。它将配合 autowire 属性使用，下面给出了配置示例。

```
@Configurable(autowire=Autowire.BY_NAME, dependencyCheck=true)
```

最后，看看 preConstruction 属性。它能够控制调用领域对象的构建器前是否注入协作者，比如下面展示了这一属性的使用，preConstruction 属性取值为 false，即 SecurityAccount()构建器会被先触发，随后才是 setBankSecurityService()方法，因此这段代码的执行不会成功。

```
@Configurable(autowire=Autowire.BY_TYPE, preConstruction=false)
public class SecurityAccount {

    private IBankSecurityService bankSecurityService;

    public SecurityAccount() {
        Assert.notNull(this.bankSecurityService);
    }

    public void setBankSecurityService(IBankSecurityService bankSecurityService) {
        this.bankSecurityService = bankSecurityService;
    }

...
```

3.6.3 通过META-INF/aop.xml（或aop-ajc.xml）控制启用的切面集合

当前，存储 AnnotationBeanConfigurerAspect 切面的 JAR 包中还持有其它 AspectJ 切面。为控制启用的切面集合，开发者需要使用 META-INF/aop.xml 或 aop-ajc.xml 配置文件。下面给出了一配置示例。

```
<!DOCTYPE aspectj PUBLIC
 "-//AspectJ//DTD//EN" "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">

<aspectj>
  <weaver
     options="-showWeaveInfo -XmessageHandlerClass:org.springframework
             .aop.aspectj.AspectJWeaverMessageHandler">
        <include within="com..*"/>
  </weaver>
  <aspects>
        <include within="org.springframework.beans.factory
             .aspectj.AnnotationBeanConfigurerAspect"/>
  </aspects>
</aspectj>
```

AspectJWeaverMessageHandler 监听了 AspectJ 6 产生的各种织入事件，并输出到控制台

上。<aspects/>元素中的<include/>子元素指定了待启用的切面。通过 aspectj.dtd，开发者能够获得这一 DTD 的完整介绍。

3.6.4 <context:spring-configured/>元素

借助<context:spring-configured/>元素能起到隐式使用 AnnotationBeanConfigurerAspect 切面的目的，下面给出了配置示例（springconfigured.xml）。

```
<context:spring-configured/>

<bean scope="prototype" autowire="byType"
    class="com.openv.spring3x.aop.SecurityAccount"/>

<bean class="com.openv.spring3x.aop.BankSecurityServiceImpl"/>
```

同样地，运行 ConfigurableDemo 时需要指定上述一致的-javaagent JVM 参数。

3.6.5 初探<context:load-time-weaver/>元素

除了<context:spring-configured/>元素能够隐式使用 AnnotationBeanConfigurerAspect 切面外，<context:load-time-weaver/>元素也能。比如，ltw.xml 中配置了如下内容。

```
<context:load-time-weaver/>

<bean scope="prototype" autowire="byType"
    class="com.openv.spring3x.aop.SecurityAccount"/>

<bean class="com.openv.spring3x.aop.BankSecurityServiceImpl"/>
```

此时，开发者需要指定如下 JVM 参数，否则 AspectJ LTW 行为无法激活。

```
-javaagent:../../lib/org.springframework.instrument-3.0.1.RELEASE-A.jar
```

事实上，<context:load-time-weaver/>只是兼容<context:spring-configured/>，它的功能非常强大，我们将在相关章节深入阐述<context:load-time-weaver/>的机理及使用。

3.7 小结

借助 AOP 及 Spring AOP 技术，我们能够控制“入侵性”，并且将各种企业级服务优雅地引入到企业应用的研发工作中。当 Spring AOP 技术实现不能够完全满足企业应用的需求时，开发者可启用 AspectJ 集成支持，从而满足各种苛刻的企业计算需求。本章内容对这些

内容进行了全面而深入的探讨。

本书后续内容将逐步进入到 Spring 内置的 Java EE 服务抽象及集成中，先从 DAO 层集成支持开始吧！

4 DAO层集成支持

4.1 RDBMS持久化操作抽象支持

4.2 JDBC集成支持

4.2.1 JDBC最佳实践

4.3 事务集成支持

4.4 集成测试支持

4.5 在AspectJ 6 应用中启用@Transactional注解

4.6 小结

5 Hibernate、JPA集成

5.1 Hibernate集成支持

5.2 JPA集成支持

5.3 智能处理Java EE容器中的装载期织入（LTW）

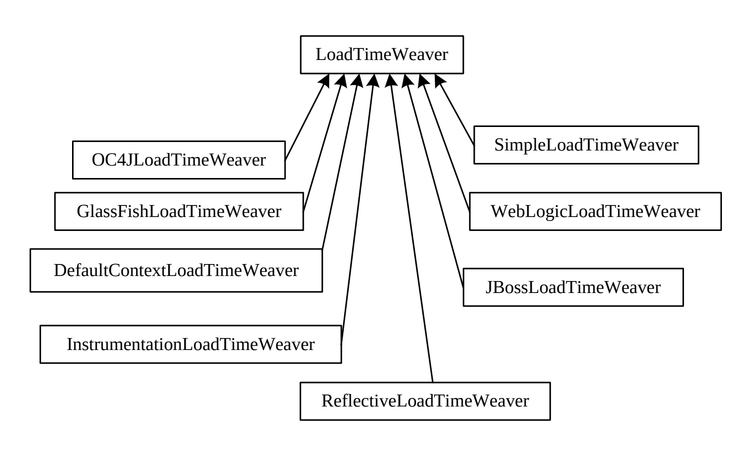


图 5-1　LoadTimeWeaver 继承链

5.4 小结

6　O/X Mapping集成支持

现如今，O/X Mapping 的使用非常频繁，即对象与 XML 间的映射工作。Spring 3.0 内置的 O/X Mapping 集成支持是基于 Spring Web Services 项目开发而成的，它们对主流的 O/X Mapping 提供者进行了封装、集成。接下来仔细研究它们的使用及内部构造。

6.1　O/X Mapping集成支持

所谓 O/X Mapping，即 Java Object 与 XML 文档间的映射工作，这是类似于 O/R Mapping 的东东，只不过 Hibernate 类似产品的目标是关系数据库，而 O/X Mapping 提供者的目标是 XML 文档。在 O/X Mapping 中，主要存在两种对象，Marshaller 和 Unmarshaller，前者负责完成对象到 XML 的转换映射工作，而后者完成 XML 到对象的转换映射工作，它们共同完成各种复杂 O/X Mapping 工作。

就目前而言，主要存在 XMLBeans（http://xmlbeans.apache.org/）、JAXB 2.0、Castor（http://castor.org/）、JiBX（http://jibx.sourceforge.net/）、XStream（http://xstream.codehaus.org/）等 O/X Mapping 提供者。在 Web 服务及 XML 使用较频繁的场合经常会使用到它们。其中，JAXB 2.0 是 JCP 规范（JSR222）；而 XMLBeans 功能非常强大、使用面也广，对 XML Schema 提供了最全面的支持。开发者要根据自身的需要灵活选用它们，毕竟不同 O/X Mapping 提供者的定位、易用程度不一样。

Spring 于 org.springframework.oxm 包内置了 O/X Mapping 集成支持，图 6-1 展示了其中的主要内容。这一集成支持对主流 O/X Mapping 提供者进行了统一的抽象、封装工作。

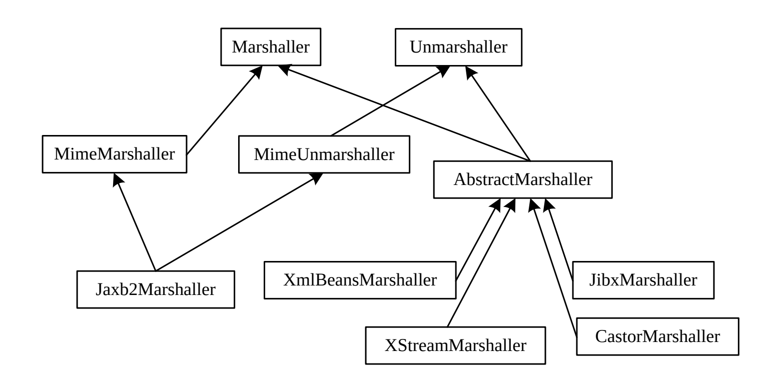


图 6-1　Marshaller 及 Unmarshaller 继承链

在这一继承链中，Marshaller 和 Unmarshaller 是最为基础的两个接口，下面大致研究一

下它们的构成情况。

6.1.1 Marshaller及Unmarshaller接口

Marshaller 接口的定义摘录如下。通过调用其 marshal()方法，能够完成对象到 XML 的转换和映射操作。

```
package org.springframework.oxm;

import java.io.IOException;
import javax.xml.transform.Result;

public interface Marshaller {

    boolean supports(Class<?> clazz);

    void marshal(Object graph, Result result) throws IOException,
        XmlMappingException;

}
```

Unmarshaller 接口的定义摘录如下。通过调用其 unmarshal()方法，能够完成 XML 到对象的转换和映射操作。

```
package org.springframework.oxm;

import java.io.IOException;
import javax.xml.transform.Source;

public interface Unmarshaller {

    boolean supports(Class<?> clazz);

    Object unmarshal(Source source) throws IOException,
        XmlMappingException;

}
```

物理上，Spring 内置了 5 种不同的 Marshaller 和 Unmarshaller 实现类，具体参考图 6-1。接下来，我们将结合 Eclipse oxmdemo 项目阐述它们的具体使用。

6.2 实践XMLBeans集成支持

本节以 XMLBeans 提供者为例阐述 O/X Mapping 集成支持的使用。有关 XMLBeans 的

背景知识，开发者可参考其官方网站。

6.2.1 借助Ant生成XMLBeans JAR

在使用 XMLBeans 期间，开发者需要准备好一 XML Schema 文件（XSD），这是类似 SQL DDL 的元数据语言。下面摘录了 stock.xsd XSD 文档的内容。XSD 对 XML 文档的语法和语义进行了建模。

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="qualified"
    targetNamespace="http://www.open-v.com/spring3x/xmlbeans"
    xmlns:tns="http://www.open-v.com/spring3x/xmlbeans">

    <element name="StockInfo">
        <complexType>
            <sequence>
                <element name="name" type="string" minOccurs="1"/>
                <element name="code" type="int" minOccurs="1"/>
                <element name="desc" type="string" />
                <element name="price" type="double" minOccurs="1"/>
            </sequence>
        </complexType>
    </element>

</schema>
```

借助如下类似 Ant build.xml 配置文件能够生成上述 XSD 对应的 XMLBeans JAR 包。

```
<?xml version="1.0" encoding="GBK"?>

<project name="xmlbeanstools" basedir=".">

    <path id="all-libs">
        <fileset dir="../../lib" includes="*.jar"/>
    </path>

    <target name="stock">
       <taskdef name="xmlbean"
            classname="org.apache.xmlbeans.impl.tool.XMLBean"
            classpathref="all-libs"/>
      <xmlbean schema="xmlbeans" srcgendir="gensrc/xmlbeanstools/"
            destfile="lib/stock.jar" classpathref="all-libs"/>
   </target>

</project>
```

好了，stock.jar 被成功创建的同时，对应的源码也被保留在 gensrc 目录的 xmlbeanstools

子目录中，有兴趣的开发者可以去研究一下。

6.2.2 XmlBeansMarshaller实现类

XmlBeansMarshaller 简化了 XMLBeans 的使用，下面摘录了 xmlbeans.xml 配置文件中定义的 XmlBeansMarshaller 及其协作者。XmlOptionsFactoryBean 用于快速创建 XmlOptions 对象，并自定义各种 O/X Mapping 行为。或者，开发者可以不用提供 XmlOptionsFactoryBean，即使用默认配置。

```
<bean id="xmlBeansMarshaller"
    class="org.springframework.oxm.xmlbeans.XmlBeansMarshaller">
    <property name="xmlOptions" ref="xmlOptionsFactoryBean"/>
</bean>

<bean id="xmlOptionsFactoryBean"
    class="org.springframework.oxm.xmlbeans.XmlOptionsFactoryBean">
    <property name="options">
        <props>
            <prop key="SAVE_PRETTY_PRINT">true</prop>
            <prop key="CHARACTER_ENCODING">UTF-8</prop>
        </props>
    </property>
</bean>
```

下面摘录了 OxmXmlbeansDemo 示例应用，其展示了 XmlBeansMarshaller 的使用。其中，上述 XMLBeans JAR 包持有的 StockInfoDocument 和 StockInfo 接口反映了 stock.xsd 中定义的<element name="StockInfo"/>元素。通过操作它们，应用便在内存中创建了吻合这一 XML Schema 定义的 XML 文档，如果需要，可以将这一文档保存到硬盘中，比如 stock.xml 中。

```
StockInfoDocument stockInfoDocument = StockInfoDocument.Factory.newInstance();
StockInfoDocument.StockInfo stockInfo = stockInfoDocument.addNewStockInfo();
stockInfo.setName("OWCY");
stockInfo.setDesc("OWCY Inc.");
stockInfo.setCode(168000);
stockInfo.setPrice(100.0);

XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
FileWriter writer = new FileWriter("stock.xml");
XMLStreamWriter streamWriter = outputFactory.createXMLStreamWriter(writer);
Result result = StaxUtils.createStaxResult(streamWriter);

marshaller.marshal(stockInfoDocument, result);

writer.flush();
```

借助 Marshaller 的 marshal()方法，我们能够将 Java 对象转换并映射成 XML 文档，并保存到 stock.xml 中。此时，开发者不用再借助各种底层 XML API 处理 XML 文档本身了，这正是 O/X Mapping 的优势。这类似于使用 Hibernate 期间，借助 Hibernate Session，开发者能够将 Java 对象转换成 RDBMS 记录，而不用借助 SQL DML 语句与数据库进行交互。

6.2.3 <oxm:xmlbeans-marshaller/>元素

如果借助 Spring 内置的<oxm/>命名空间，则上述 xmlbeans.xml 配置文件可以得到简化，比如下面给出了配置示例，它将代替 XmlBeansMarshaller 的定义，即开发者不用再显式定义 XmlBeansMarshaller 对象了。

```
<oxm:xmlbeans-marshaller id="xmlBeansMarshaller"
    options="xmlOptionsFactoryBean"/>
```

类似地，针对 JAXB 2.0 和 JiBX 提供者，开发者可启用<oxm:jaxb2-marshaller/>、<oxm:jibx-marshaller/>元素。

有关 O/X Mapping 集成支持的更多使用场合

通常，Web 服务离不开处理各类 XML 工件，比如 SOAP 消息。Spring Web Services 到处深入使用到 O/X Mapping 集成支持。本书附录 C 详细阐述了 O/X Mapping 集成支持在 Web 服务中的应用。

6.3 小结

Spring 内置的 O/X Mapping 集成支持非常不错，它不仅有效屏蔽了各 O/X Mapping 实现的差异性，而且还提供了统一、高效的编程模型供开发者使用。开发者可以在各种应用中使用它。除此之外，在 Spring 内部，这一集成支持也被广泛使用，比如 JMS 集成支持中的 MarshallingMessageConverter（位于 org.springframework.jms.support.converter 包）、Spring Web MVC 中的 MarshallingView 和 MarshallingHttpMessageConverter 等。

下章内容将进入到 Spring 集成的其它各种 Java EE 容器服务中。

7 集成Java EE其他容器服务

7.1 简化JNDI操作

7.2 集成EJB 3.1

7.3 线程池及任务调度集成支持

7.4 集成JMS

7.5 集成JavaMail

7.6 集成分布式操作

7.7 集成JMX

7.8 集成Java EE连接器架构

7.9 小结

8 Web层集成支持

8.1 Spring Web MVC框架

8.2 Spring Portlet MVC框架

8.3 REST架构风格

8.4 小结

9 高级Spring 3.0 特性

9.1 优雅销毁DI容器

9.2 小结

10 附录A：安装及使用SpringSource Tool Suite

STS 是 SpringSource 公司（VMWare 子公司）针对 Spring 开发者研发的一款重量级、免费的开发工具。本附录将围绕 STS（SpringSource Tool Suite，SpringSource 工具套件）的安装及使用展开论述。

10.1 获得SpringSource Tool Suite

SpringSource 公司于 http://www.springsource.com/products/sts 网址提供了 STS 的下载入口。图 10-1 给出了对应的操作界面。

Current Version - 2.7.1.RELEASE

Description	Link	Size	Hash
Eclipse 3.7			
Windows	springsource-tool-suite-2.7.1.RELEASE-e3.7-win32-installer.exe	361MB	sha1 - md5
Windows	springsource-tool-suite-2.7.1.RELEASE-e3.7-win32.zip	360MB	sha1 - md5
Windows (64bit)	springsource-tool-suite-2.7.1.RELEASE-e3.7-win32-x86_64-installer.exe	361MB	sha1 - md5
Windows (64bit)	springsource-tool-suite-2.7.1.RELEASE-e3.7-win32-x86_64.zip	359MB	sha1 - md5
Eclipse 3.7.x			
Update Site	springsource-tool-suite-2.7.1.RELEASE-e3.7-updatesite.zip	160MB	sha1 - md5

图 10-1　下载 SpringSource Tool Suite 的操作界面

无论是 Windows，还是 Linux，还是 Apple Mac OS X 操作系统用户，他们都能够使用到 STS。这里以下载 springsource-tool-suite-2.7.1.RELEASE-e3.7-win32-installer.exe 为例。

10.2 安装SpringSource Tool Suite

安装 STS 的步骤很简单，但在此之前开发者要准备好 JDK，比如在 D:\jdk1.6.0_27 位置安装好 Java SE 6。并设置好若干环境变量，比如将 JAVA_HOME 设置成“D:\jdk1.6.0_27”、classpath 设置成“.;”、将“%JAVA_HOME%\bin”追加到 Path 环境变量中。

当通过 DOS 窗口成功运行“java -version”后，开发者便可开始进行 STS 的安装工作了。图 10-2 示意了 STS 的安装，并将它安装在 D:\springsource 位置。

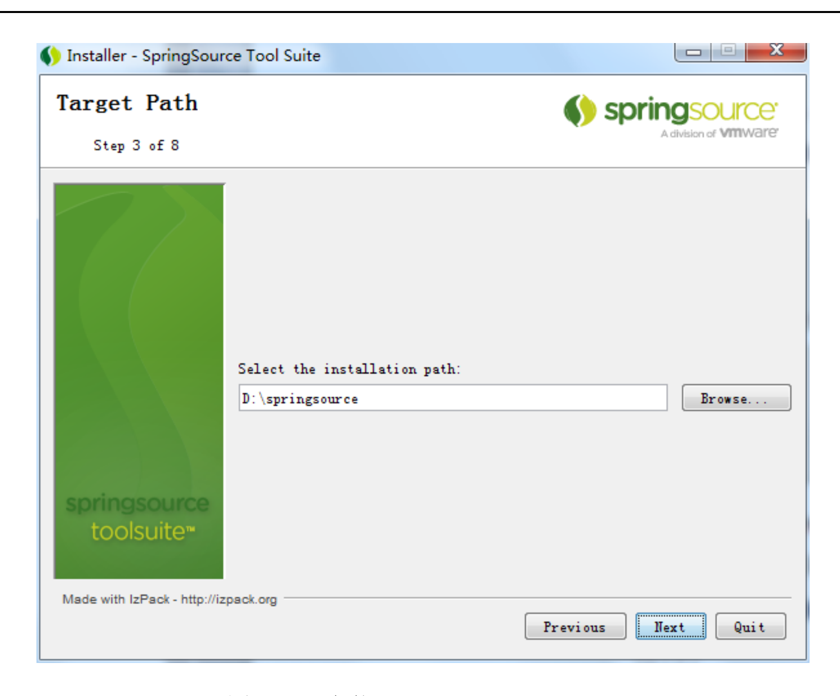


图 10-2　安装 SpringSource Tool Suite

随后，开发者需要为 STS 选定一 JDK，见图 10-3。

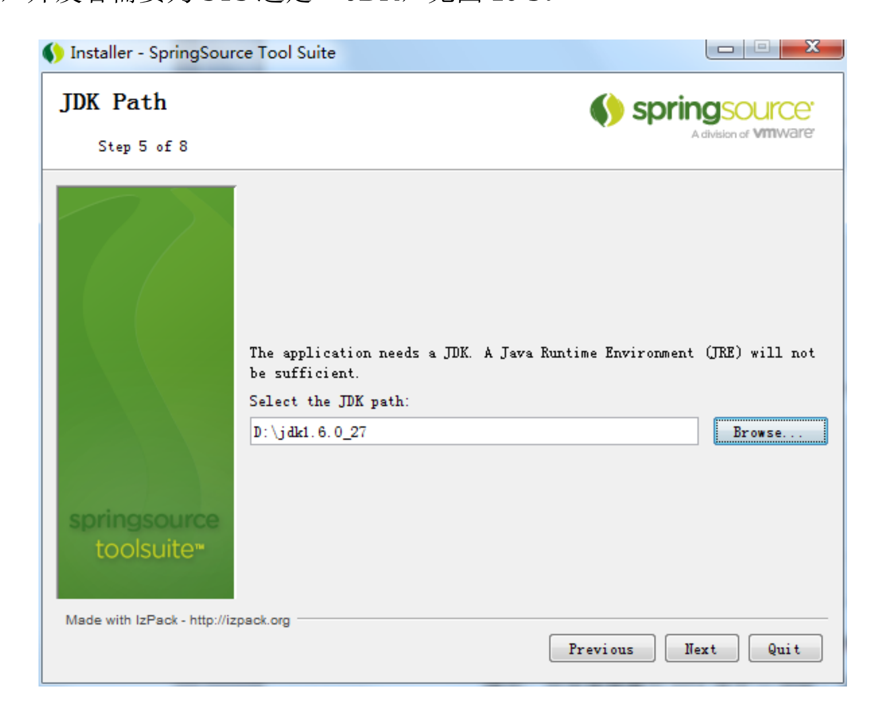


图 10-3　选择 JDK

安装完后，开发者即可启动 STS，在选定 Eclipse 工作空间（图 10-4）后，便可开始享受 STS 带来的快乐了。本书假定使用 D:\springsource\workspace 目录作为工作空间。

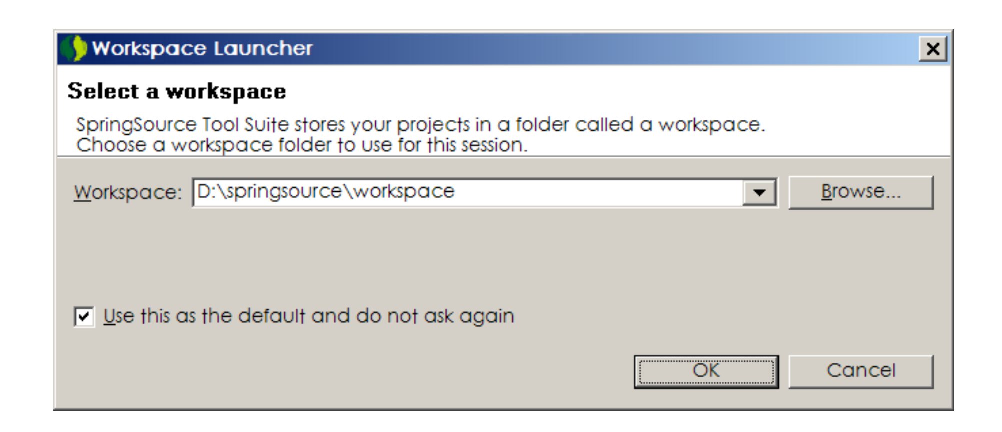


图 10-4　选择工作空间

10.3 使用SpringSource Tool Suite

本节内容将从不同角度分别阐述 STS 的使用。

10.3.1针对Spring 3.1 的支持

目标 Java SE/Java EE 项目为使用到 STS 内置的 Spring IDE 支持，则必须启用“Spring Project Nature”。图 10-5 给出了操作界面，右键单击项目能够看到它。



图 10-5　启用 Spring Project Nature 支持

随后，开发者需要打开“Spring Explorer”视图，并激活图 10-6 界面。通过它，能够完成多项设置，比如 Spring DI 配置文件的设定、配置集合、命名空间的启用等。

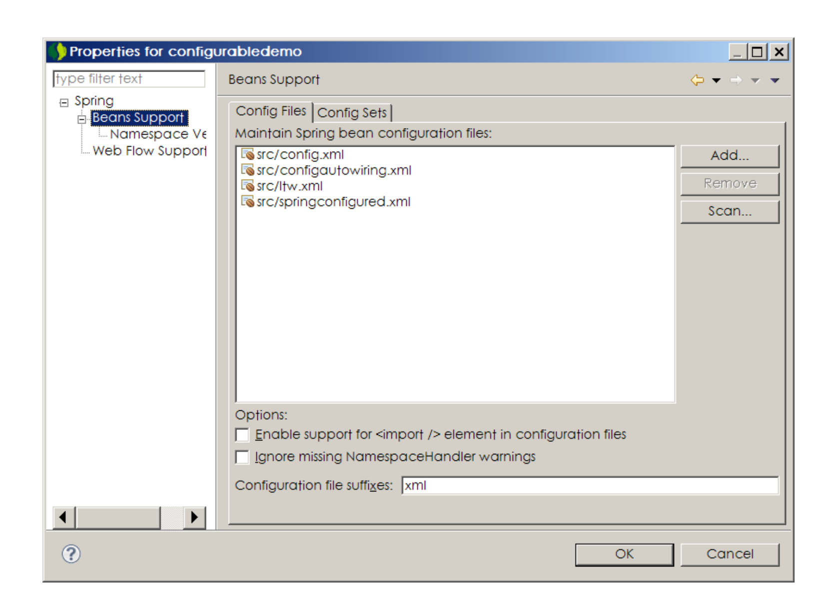


图 10-6　配置项目中同 Spring 相关的内容

当我们将若干 Spring DI 配置文件放置到“Config Files”Tab 页后，便能够通过“Spring Explorer”视图浏览到它们，并以可视化方式展现出 DI 容器的内部结构，比如各协作者间的层次关系。图 10-7 展示了相关界面。

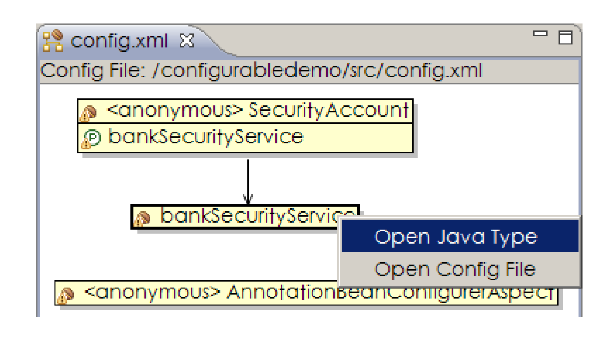


图 10-7　可视化 DI 容器的内部结构

图 10-7 展示的内容非常重要。实践证明，开发者之间经常需要探讨应用的设计、架构师需要评审开发者的工作，而可视化 DI 容器无疑为这些工作提供了最好的帮助。而且，开发者可以同这一可视化 DI 容器进行交互。

除了针对 DI 容器进行可视化操作外，开发者还可以借助 Spring IDE 内置的 XML 辅助编写能力，以加快 Spring DI 配置文件的编写速度及质量的保证，因为一旦有问题，Spring IDE 会及时指出。为启用 XML 辅助编写，我们需要用“Spring Config Editor”打开 Spring DI 配置文件。图 10-8 展示了这一特性的使用，通常用 Alt+“/”键组合能够激活代码辅助特性。

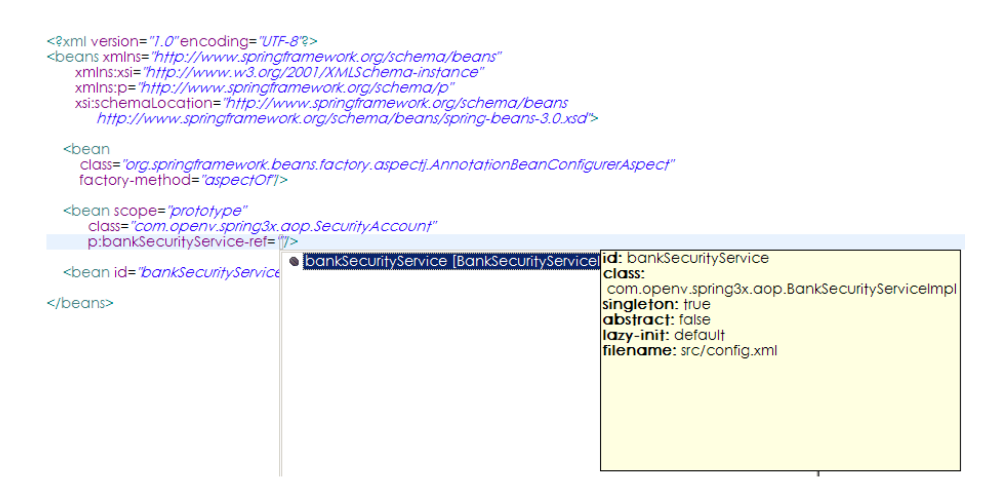


图 10-8　Spring IDE 内置的 XML 辅助编写能力

另外，Spring IDE 还有其它许多有意思的地方，比如图 10-9 展示了开发者可以通过可视化方式控制当前 Spring DI 配置文件启用的命名空间集合。

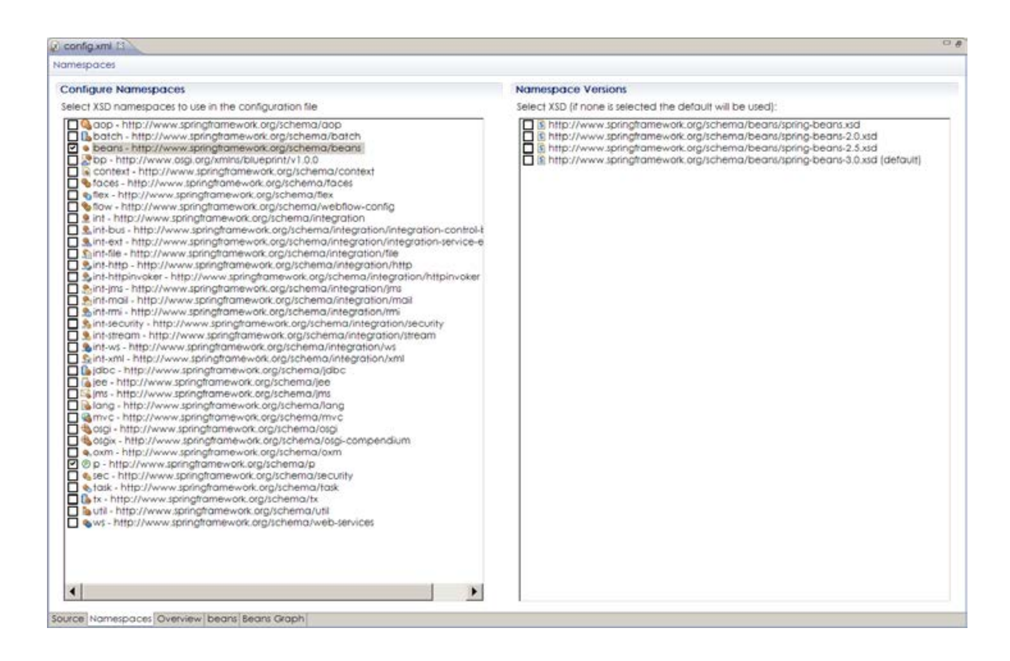


图 10-9　控制当前 Spring DI 配置文件启用的命名空间集合

更多细节，还需要开发者在日常开发工作期间体会、总结。

10.3.2针对Spring Web Flow的支持

10.3.3针对Spring Batch的支持

10.3.4针对Spring Roo的支持

Spring Tools ▸ Add Roo Project Nature

图 10-10　目标项目启用 Roo IDE 支持

11 附录B：Spring 3.1 内置的命名空间

11.1 <beans/>命名空间

11.2 <context/>命名空间

11.3 <util/>命名空间

12 附录C：Spring Web Services

12.1 文档驱动的Web服务

12.2 面向OXM的Web服务实现策略

12.3 Web服务安全

13 附录D：Spring Web Flow

13.1 流程致胜

13.2 探索Spring Web Flow

14 附录E：Spring BlazeDS Integration

14.1 Flex－RIA王者

14.2 简化BlazeDS的使用

14.3 深入到Spring BlazeDS Integration中

15 附录F：Spring Roo

15.1 快速研发之道

15.1.1 Spring Roo概述

15.2 Spring Roo架构哲学

15.3 深入到Spring Roo中

16 附录G：相关资料

16.1 图书

- 《Expert One-on-One J2EE Development without EJB》，WILEY。作者，Rod Johnson、Juergen Hoeller。2004.6
- 《Professional Java Development with the Spring Framework》，WILEY。作者，Rod Johnson、Juergen Hoeller 等。2005.7
- 《精通 Spring—深入 Java EE 开发核心技术》，第三版，电子工业出版社。作者，罗时飞。2008.10
- 《AspectJ in Action: Enterprise AOP with Spring Applications》，第二版，Manning。作者，Ramnivas Laddad。2009.9

16.2 网站

- Spring 官方网站：http://www.springsource.org/
- Spring 社区下载：http://www.springsource.com/download/community。通过这一位置，开发者能够下载到各种 Spring 开源项目包
- Spring 社区论坛：http://forum.springsource.org/
- SpringSource 团队博客：http://blog.springsource.com/。持续跟进 Spring 的绝佳去处
- Java 官方网站：http://java.sun.com/